Implementing queries and updates on universal scheme interfaces

C. Lecluse, N. Spyratos

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 805

IMPLEMENTING QUERIES AND UPDATES ON UNIVERSAL SCHEME INTERFACES

Christophe. LECLUSE
Nicolas. SPYRATOS

MARS 1988

IMPLEMENTING QUERIES AND UPDATES

ON UNIVERSAL SCHEME INTERFACES

C. LECLUSE and N. SPYRATOS

Université de Paris Sud

91405 Orsay cedex

FRANCE

ABSTRACT Using partition semantics [LS87], we show that to every relational universe U and set of functional dependencies F, there corresponds a unique database scheme (called the canonical scheme) such that:

(1) Every query on the universe can be answered uniquely by a relational expression on the canonical scheme

(2) Every update of the universal relation can be translated uniquely into a transaction on the canonical scheme.

Our results render the relational model logically independent with respect to both queries *and updates*, thus subsuming previous approaches to the problem [MRSSW87].

REALISATION DES REQUETES ET DES MISES A JOUR DANS UNE INTERFACE RELATION UNIVERSELLE

Christophe LECLUSE et Nicolas SPYRATOS

Université de Paris Sud

91405 Orsay cedex

FRANCE

RESUME Nous montrons, en utilisant la sémantique du modèle de partition [LS87] , qu'à chaque univers relationnel et ensemble de dépendances fonctionnelles, on peut associer un schémas relationnel (appelé schémas canonique) tel que:

(1) Chaque requête peut être transformée en une requête relationnelle sur le schémas canonique, et

(2) Chaque mise-à-jour sur l'interface relation universelle peut être transformée en une mise à jour sur le schemas canonique.

Ces résultats donnent au modèle relationnel l'indépendance logique tant au niveau des requêtes *que des mises à jour* . Ils recouvrent ainsi les précédentes approches du problème [MRSSW87].

1. INTRODUCTION

As noted by Maier,Vardi and Ullman [MVU84], the relational data model has gone far toward *physical* data independence, but has not achieved the goal of *logical* data independence. That is, users of relational systems are relieved of specifying access paths within the structure of a single relation, but they still must navigate between relations. Users and application programs are protected from changes in the physical implementation of relations, but not from changes in the logical structure of a database, such as decomposition made for normalization or efficiency reasons.

Universal scheme interfaces are an attempt to achieve logical data independence. In a universal scheme interface, all the semantics of the database is loaded onto the attributes. Queries are phrased in terms of attributes alone and the user does not need to know which attributes are in which relations. That is, in a universal scheme interface, a database is presented as a semantic whole, accessible through its attributes alone.

The idea that data can, at least in principle, be thought of as residing in a single relation is an intuitively appealing one. In fact, even in the normalization process, the implicit assumption is that it makes sense to talk about attributes disembodied from any particular relation scheme and, therefore, with a meaning of their own. However, what this meaning should be was not clear, and this gave rise to much controversy. The breakthrough was Mendelzon's "weak" or "representative" instance view of universal relations [M84] exploited by Sagiv [S83] in a real system. As with dependencies, there are in fact many sound ways one can view the universal relation; Maier, Rozenshtein and Warren [MRW86] is a key paper integrating these differing viewpoints. They also survey a large number of systems that are, implicitely or explicitely, based on a universal scheme interface.

The weak instance view of universal relations seems to provide the right framework for querying universal scheme interfaces. However, there does not seem to be the flexibility to update over arbitrary schemes that there is to query over arbitrary schemes. Proposals to remedy this defficiency (such as introducing "missing value" nulls) have led to increased complexity without really solving the update problem.

In this paper, we propose a novel approach for processing queries <u>and</u> updates in a universal scheme interface, based on the set-theoretic semantics of the partition model [S84,CKS86]. In our approach, the implementation of a universal scheme interface is done in two parts as follows:

DESIGN PHASE :

Given universe U and set of functional dependencies F,

(1) We associate U and F with a (uniquely defined) database scheme, called the *canonical scheme* of U and F.

(2) We associate every nonempty subset Q of U with three (uniquely defined) objects:

- A relational expression on the canonical scheme
- An insertion transaction on the canonical scheme
- A deletion transaction on the canonical scheme.

Note: By transaction, we mean a sequence of queries, insertions and deletions.

RUN TIME :

The data is stored according to the canonical scheme. User queries and updates are expressed in terms of attributes alone, and the user does not have to know which attributes are in which relations of the canonical scheme. A query or update, expressed in terms of a set of attributes Q, is processed as follows:

query: The relational expression associated, during the design phase, with Q is evaluated on the current database.

insertion: The insertion transaction associated, during the design phase, with Q is evoked and executed on the current database.

deletion: The deletion transaction associated, during the design phase, with Q is evoked and executed on the current database.

It is important to note that, following our approach, the overhead required for the implementation of a universal scheme interface, is a one-shot operation. Indeed, this overhead appears only during the design phase, when one must generate the canonical scheme, and the relational expression and transactions associated with each distinct set of attributes. During run-time, however, no particular overhead is required, and the processing of queries and updates is done using traditional DBMS techniques. Therefore, our approach of implementing a universal scheme interface can be seen as adding an upper layer on a traditional DBMS. This upper layer is currently being developped in our laboratory.

The paper is organized as follows. In Section 2, we recall the basic definitions from the partition model [LS87] which provides the underlying semantics of our approach. In Section 3, we define partition semantics for queries *and updates* in a universal scheme interface; we show that, in terms of queries, our semantics is equivalent to weak instance semantics. In Section 4, we show how partition semantics can lead to unambiguous semantics for universal scheme interfaces. Section 5 contains some conclusions and suggestions for further work.

2. PARTITION SEMANTICS.

2.1 Informal overview.

Consider the following database containing only two tuples

AGE	SITUATION
Young	*Unemployed*

SITUATION	SEX
Unemployed	*Female*

The tuple *Young Unemployed* can be seen as a string of two uninterpreted symbols, *Young* and *Unemployed*. Now, think of a possible world, and let Ω be the set of all individuals in that world. Moreover, let I(*Young*) be the set of all individuals of Ω that are young, and call I(*Young*) the interpretation of *Young*. Similarly, let I(*Unemployed*) be the set of all individuals of Ω that are unemployed and call I(*Unemployed*) the interpretation of *Unemployed*. Clearly, the intersection I(*Young*)∩I(*Unemployed*) is the set of all individuals of Ω that are both young and unemployed. It is precisely this intersection that we define to be the interpretation of the tuple *Young Unemployed*. That is,

$$I(\mathit{Young}\ \ \mathit{Unemployed}) = I(\mathit{Young}) \cap I(\mathit{Unemployed}).$$

In other words, the interpretation of a tuple is the intersection of the interpretations of its constituent symbols.

This kind of set-theoretic semantics of tuples allows for a very intuitive notion of truth. A tuple t is called <u>true</u> in interpretation I iff I(t) is nonempty. Thus, for example, the (atomic) tuple *Young* is true iff I(*Young*) is nonempty, that is iff there is an individual in Ω which is young (at least one such individual). Similarly, the tuple *Unemployed* is true in I iff there is an individual in Ω which is unemployed. Finally, the tuple

Young Unemployed is true in I iff there is an individual in Ω which is both young and unemployed.

The set-theoretic semantics just introduced allow for a very natural notion of inference through set-containment. To see this, consider the following question:

Assuming that

Young Unemployed is true in I

and that

Unemployed Female is true in I

can we infer that

Young Unemployed Female is true in I

If we recall the definition of truth given earlier, then we can reformulate this question as follows:

Assuming that

$$I(Young) \cap I(Unemployed) \neq \emptyset \qquad (1)$$

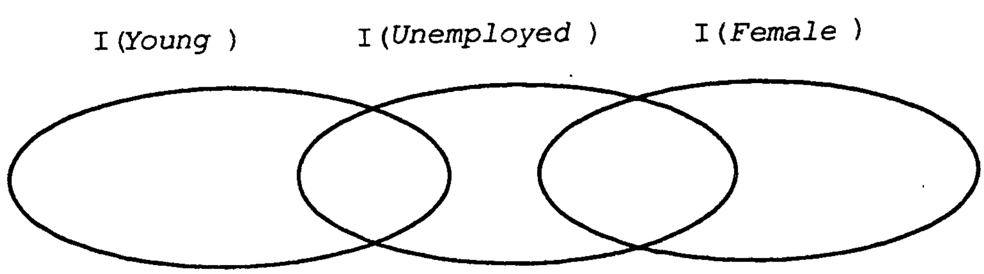
and that

$$I(Unemployed) \cap I(Female) \neq \emptyset \qquad (2)$$

can we infer that

$$I(Young) \cap I(Unemployed) \cap I(Female) \neq \emptyset \qquad (3)$$

Clearly the answer depends on the interpretation I, and the following diagram shows a case where the answer is no.



However, if we impose constraints on the interpretation I, for instance, if we require that

$$I(Unemployed) \subseteq I(Young) \qquad (4)$$

(meaning that all unemployed individuals are young)

or that

$$I(Unemployed) \subseteq I(Female) \qquad (5)$$

(meaning that all unemployed individuals are female)

then the answer is yes. Roughly speaking, we can summarize our example as follows:

[(1) and (2)] does <u>not</u> imply (3)

whereas

[(1) and (2)] and [(4) or (5)] implies (3).

It turns out that constraints such as (4) and (5) provide the right interpretation for functional dependencies. (In fact, (4) interprets SITUATION→AGE, (5) interprets SITUATION→SEX, and we have just interpreted the fact that if SITUATION→AGE holds or SITUATION→SEX holds then the join SITUATION AGE⋈SITUATION SEX is lossless).

We shall see later that these notions of truth and inference provide the basic tools for processing queries <u>and</u> updates in a universal scheme interface. First, let us describe formally our model.

2.2 The Model.

We shall consider, separately the syntax and the semantics of our model. The syntactic part is essentially the relational model. The semantic part is a formalization of the concepts explained above.

2.2.1 Syntax

We begin with a finite, nonempty set $U=\{A_1,...,A_n\}$. The set U is called the *universe* and the A_i's are called the *attributes*. Each attribute A_i is associated with a countably infinite set of symbols (or values) called the *domain* of A_i and denoted by $dom(A_i)$. We assume that $U \cap dom(A_i)$ is empty for all i, and that $dom(A_i) \cap dom(A_j) = \emptyset$ for $i \neq j$. A *relation scheme* over U is a nonempty subset of U; we call sch(U) the set of all realtion schemes over U and we denote a relation scheme by the juxtaposition of its attributes (in any order). A *tuple* t over a relation scheme R is a function defined on R such that $t(A_i)$ is in $dom(A_i)$, for all A_i in R. We denote by dom(R) the set of all tuples over R. Clearly, dom(R) is the cartesian product of the domains of all the attributes in R. If t is a tuple over $R=A_1...A_n$, and if $t(A_j)=a_j$,

$j=1,\ldots,n$, then we denote the tuple t by $a_1 \ldots a_n$. A *relation* over R is a set of tuples over R.

Definition 2.1 A database over U is a pair (δ,F) such that

(1) δ is a function assigning to every relation scheme R over U a finite relation over R, and

(2) F is a set of ordered pairs (X,Y) such that X and Y are subsets of U.

Every pair (X,Y) is called a *functional dependency* and is denoted by $X \rightarrow Y$ ♦

Example 2.1 Consider a universe of three attributes, say $U=\{A,B,C\}$, and let $dom(A)=\{a_1,a_2,\ldots\}$, $dom(B)=\{b_1,b_2,\ldots\}$, $dom(C)=\{c_1,c_2,\ldots\}$.

Define a function δ on relation schemes over U as follows:

$\delta(AB) = \{a_1b_1, a_2b_1\}$

$\delta(BC) = \{b_1c_1, b_1c_2\}$

$\delta(R) = \varnothing$ for all R different than AB and BC.

Let F be the set $\{A \rightarrow B, BC \rightarrow A\}$. The database (δ,F) just defined is shown in Figure 2.1(b). The function δ is represented by two tables. The convention used here is that relation schemes that are assigned empty relations under δ are not represented. ♦

(a) $U = \{A,B,C\}$, $dom(A) = \{a_1,a_2,\ldots\}$, $dom(B) = \{b_1,b_2\ldots\}$, $dom(C)=\{c_1,c_2,\ldots\}$.

I $\quad a_1 \rightarrow \{1,3\} \quad a_2 \rightarrow \{2,4\} \quad b_1 \rightarrow \{1,2,3,4\}$

$\quad b_2 \rightarrow \{5,6\} \quad c_1 \rightarrow \{3\} \quad c_2 \rightarrow \{1\}$

$\quad x \rightarrow \varnothing$, for every x different than a_1,b_1,b_2,c_1,c_2.

(b) δ

AB	BC
a_1b_1	b_1c_1
a_2b_1	b_1c_2

F

$A \rightarrow B$

$BC \rightarrow A$

(c)

$I(a_1b_1) = I(a_1) \cap I(b_1) = \{1,3\}$ $\quad I(a_2b_1) = I(a_1) \cap I(b_1) = \{2,4\}$

$I(b_1c_1) = I(b_1) \cap I(c_1) = \{3\}$ $\quad I(b_1c_2) = I(b_1) \cap I(c_2) = \{1\}$

$I(a_1b_1c_1) = I(a_1) \cap I(b_1) \cap I(c_1) = \{3\}$ $\quad I(a_1b_1c_2) = I(a_1) \cap I(b_1) \cap I(c_2) = \{1\}$

FIGURE 2.1 An interpretation of U and a database d over U for which I is a model.

Given a database $D=(\delta,F)$, we call *scheme* of D, denoted by sch(D), the set of all relation schemes that are assigned non-empty relations under δ. That is, $sch(D) = \{R \in sch(U) \mid \delta(R) \neq \emptyset\}$. Thus, the database scheme in Example 2.1 consists of the relation schemes AB and BC.

2.2.2 Semantics.

We assume that the "real world" consists of a countably infinite set of objects, and we identify these objects with the positive integers. Let ω be the set of all positive integers, and let $2^{\omega} = \{ \tau / \tau \subseteq \omega\}$ be the set of all subsets of ω. The set 2^{ω} is the semantic domain in which tuples and dependencies will receive their interpretations.

Throughout our discussions, we consider fixed the universe of attributes $U = \{A_1, A_2, \ldots A_n\}$, and the associated domains $dom(A_i)$. For notational convenience, we denote by SYMBOLS the union of all attribute domains and by TUPLES the union of all domains. That is:

$$SYMBOLS = \cup_{A \in U} \; dom(A),$$

$$TUPLES = \cup_{R \in sch(U)} \; dom(R).$$

Clearly, SYMBOLS is a subset of TUPLES.

Definition 2.2 An *interpretation* of U is a function I from SYMBOLS into 2^{ω} such that

$$\forall A \in U, \; \forall a,a' \in dom(A), \; (a \neq a' \Rightarrow I(a) \cap I(a') = \emptyset) \qquad \blacklozenge$$

Thus the basic property of an interpretation is that *different* symbols of the *same* domain are assigned disjoint sets of integers. In Figure 2.1(a) we see a function I satisfying this property. The intuitive motivation behind this definition is that an attribute value, say a, is an (atomic) property, and I(a) is a set of objects having property a under I. Furthermore, an object cannot have two different properties a, a' of the same "type"; hence $I(a) \cap I(a') = \emptyset$. Given an interpretation I, we can extend it from SYMBOLS to TUPLES as follows:

$$\forall R \in sch(U), \; \forall a_1 a_2 \ldots a_n \in dom(R), \; I(a_1 a_2 \ldots a_n) = I(a_1) \cap \ldots \cap I(a_n)$$

In Figure 2.1(c), we see some example of computations of tuple interpretations. The intuitive motivation for this extension is that a tuple, say ab, is the conjunction of the (atomic) properties a and b. Accordingly, I(ab) is the set of objects having both properties a and b, and therefore, $I(ab) = I(a) \cap I(b)$. Our definition of an interpretation suggests intuitive notions of thruth and refinement for tuples, as follows:

Definition 2.3 Let I be an interpretation and let s,t be any tuples in TUPLES. We say that tuple t is *true* in I iff $I(t) \neq \emptyset$, and we say that tuple s *refines* tuple t in I iff $I(s) \subseteq I(t)$. ♦

That is, t is true in I if there is at least one individual having property t, and s refines t in I if every individual having property s (under I) also has property t. Note that every tuple t refines all its subtuples (for example, ab refines both a and b). We now define when an interpretation I is called a model of a database D.

Definition 2.4 Let $D = (\delta,F)$ be a database over U. An interpretation I of U is called a *model* of D if

(1) $\forall R \in sch(U), \ \forall t \in \delta(R), \ I(t) \neq \emptyset$.

(2) $\forall X \rightarrow Y \in F, \ \forall x \in dom(X), \ \forall y \in dom(Y), \ I(x) \cap I(y) \neq \emptyset \Rightarrow I(x) \subseteq I(y)$ ♦

Condition (1) of this definition says that every tuple appearing in the database must be true. Condition (2) says that every functional dependency must be interpreted as a function. To see this, define:

$$HEAD(I) = \{x \in dom(X) \ / \ I(x) \neq \emptyset \},$$

$$TAIL(I) = \{y \in dom(Y) \ / \ I(y) \neq \emptyset \},$$

and, for every x in HEAD(I), define

$$G(x) = y, \text{ if } I(x) \cap I(y) \neq \emptyset.$$

Now, if I verifies the functional dependency $X \rightarrow Y$ then, according to Definition 2.4, $I(x) \cap I(y) \neq \emptyset \Rightarrow I(x) \subseteq I(y)$. On the other hand, for all $y' \neq y$ in dom(Y), we have $I(y') \cap I(y) = \emptyset$. Thus, there is at most one y such that $I(x) \subseteq I(y)$. It follows that there is at most one y such that $I(x) \cap I(y) \neq \emptyset$. Thus, every x is associated with at most one y, under G. So G is a function from HEAD(I) into TAIL(I). Let us note that $X \rightarrow Y$ is still interpreted as a function even if we replace condition (2) by the stronger condition

(2') $\forall x \in dom(X),\ \exists y \in dom(Y),\ I(x) \subseteq I(y)$.

The difference between (2) and (2') is that condition (2) interprets $X \rightarrow Y$ as a partial function whereas condition (2') interprets $X \rightarrow Y$ as a total function. In this paper, we adopt the (more general) condition (2) in our definition of a model. We shall come back to this remark when discussing chase and weak chase in the following section.

Having defined the concept of model, we can now define the concept of consistency. A database D is called *consistent* iff D posseses at least one model; and otherwise D is called *inconsistent*. For example, the database D = (δ,F) of Figure 2.1 is consistent as the interpretation shown in that same figure is a model of D. On the other hand, the following database is inconsistent, as no interpretation I can verify the tuples ab and ab', and the dependency $B \rightarrow A$, all at the same time.

δ

A	B
a	b
a'	b

B	C
b	c

F

$B \rightarrow A$
$A \rightarrow C$

Definition 2.5 Let D=(δ,F) be a database over universe U. Let s and t be any tuples in TUPLES. We say that D *implies* t, denoted by $D \models t$, iff $m(t) \neq \emptyset$ for every model m of D. We say that D *implies* s≤t, denoted by $D \models t \leq s$, if $m(t) \subseteq m(s)$, for every model m of D. ♦

Clearly, D implies all tuples appearing in D. This follows immediately from the definition of a model. On the other hand, the database D of Figure 2.1 does not imply the tuples $a_2b_1c_1$ and $a_2b_1c_2$, as they are false in the model I shown in the figure.

Given a model m of D, we denote by T(m) the set of all tuples in TUPLES which are true in m. Let us denote by mod(D) the set of all models of D, and let us define

$$T(D) = \cap_{m \in mod(D)} T(m).$$

T(D) is clearly the set of all tuples implied by D. In the following section, we use the set T(D) in order to define semantics for queries and updates in a universal scheme interface.

3. SEMANTICS OF QUERIES AND UPDATES.

Our semantics for queries corresponds to the weak instance semantics, but our semantics for updates is new.

3.1 Queries.

We define queries as in a universal scheme interface. That is, a query is a set of attributes, plus a selection condition. However, in order to arrive at the formal definition of a query and its answer, we need some preliminary definitions and notations.

Given a universe U and a relation scheme Q over U, we call *elementary condition* over Q any expression of the form X=x, where X is a subset of Q and x a tuple over X.

Definition 3.1 We call *elementary query* over U, any expression of the form Q/X=x, such that Q is a relation scheme and X=x is an elementary condition over Q. Given a universe U, an elementary query Q/X=x, and a database D over U, the *answer* of Q/X=x with respect to D, denoted by α(Q/X=x,D), is defined as follows:

$$\alpha(Q/X=x,D) = \{t \in dom(Q) \,/\, D \models t,\ D \models t \leq x\} \quad \blacklozenge$$

In other words, α(Q/X=x,D) is the set of all tuples over Q implied by D and refining x in D.

Example 3.1 Consider the universe U={A,B,C,D}, and a database D = ({ab,bc,cd}, {B→D, C→D}). Let m be any model of D. As cd is in D, we have m(c) $\subseteq$ m(d) because of the dependency C→D. Hence, we have m(bcd) = m(b)$\cap$m(c)$\cap$m(d) = m(b)$\cap$m(c) = m(bc). As bc is in D, it follows that m(bcd)$\neq\emptyset$ and, thus, m(bd)$\neq\emptyset$. So D implies bd. Using ab, bd and the dependency B→D, we can show in the same way that D implies ad. So the answer to the query AD in the database D is {ad}. Notice that, without the dependencies B→D <u>and</u> C→D, the answer to the query AD would be empty!

♦

Clearly, in order to answer elementary queries, we must solve the following inference problems: Given a database D and tuples t and x,

(1) Decide whether D $\models$ t,

(2) Decide whether $D \models t \leq x$.

A procedure for solving these problems is given in [LS87].

Elementary conditions can be combined using logical connectives in order to form more complex conditions. We call *selection condition* over Q, any combination of elementary conditions over Q, using logical connectives.

Definition 3.2 We call *query over U*, any expression of the form Q/s such that Q is a relation scheme and s is a selection condition. The answer to a query over U is defined recursively, based on the answers to elementary queries, as follows:

$\alpha(Q/\neg s,D) = \alpha(Q,D) - \alpha(Q/s,D)$,

$\alpha(Q/s \wedge s',D) = \alpha(Q/s,D) \cap \alpha(Q/s',D)$,

$\alpha(Q/s \vee s',D) = \alpha(Q/s,D) \cup \alpha(Q/s',D)$. ♦

3.2 Updates.

We consider the problem of inserting and deleting a single tuple t in a database $D=(\delta,F)$. (In [LS87], we show that inserting or deleting a set of tuples is equivalent to inserting or deleting the tuples, one at a time). We assume that the set F is fixed, that is, we update δ and not F (ie, we update tuples and not dependencies). We denote by BASES(F), the set of all databases over a set F of functional dependencies. Given two databases D and D' in BASES(F), we say that D and D' are equivalent, denoted by $D \equiv D'$, iff $T(D) = T(D')$. This definition is motivated by the fact that the information carried by a database D is the set T(D), namely the set of all tuples implied by D. We are interested mainly in the information T(D), and not in the representation of this information by different (but equivalent) databases. The set of all equivalence classes of BASES(F) is denoted by BASES(F)/≡ and the class of a database D is denoted by **D**. We shall take the set T(D) to be the representative of the class **D**. We say that a class **D** is *smaller* than a class **D'**, denoted by $\mathbf{D} < \mathbf{D'}$, iff $T(D) \subseteq T(D')$. What we are updating is the information T(D), rather than the database D. In other words, we are updating equivalence classes and *not* specific databases. As we update equivalence classes, we define insertion to be a function from the cartesian product BASES(F)/≡ x TUPLES into BASES(F)/≡. This function takes as arguments an equivalence class **D** and a tuple t, and returns an equivalence class **D'**. The conditions that the equivalence class D' must satisfy are stated formally in the following definition.

Definition 3.3 Let U be a universe and let F be a fixed set of functional dependencies over U. Define *insertion* of a tuple to be a function INS, from BASES(F)/≡ x TUPLES into BASES(F)/≡ , such that, for every tuple t and every database D:

(1) INS(**D**,t) is the smallest class **D'** verifying conditions (2) and (3) below:

(2) **D** < **D'**, and

(3) **D'** ⊨ t. ♦

Note that if **D**' exists then it is unique. Indeed, **D**' is the equivalence class that corresponds to

$$\cap \ \{T(D'') \ / \ T(D) \subseteq T(D''),\ t \in T(D'')\}$$

Thus, INS is a well defined function. However, as insertion of tuples may create inconsistency with respect to F, it is clear that INS is only a partial function.

A word of explanation is in order here, concerning the requirement that INS(**D**,t) be minimal. First, observe that if INS(**D**,t) exists then there may be many different equivalence classes **D'** satisfying the conditions (2) and (3) above. Thus, in order for INS(**D**,t) to be a function, we must designate a unique class **D'** as the result of insertion. The reason why we ask for a minimal class **D'**, satisfying (2) and (3), is because we want to exclude undesirable insertions of tuples ("side effects"). For example, if D=({ab},∅) and we want to insert the tuple a'b', then any of the following non-equivalent databases satisfies conditions (2) and (3) above:

D' = ({ab,a'b'},∅), D" = ({ab,a'b',a"b"},∅), ...

Of the above databases, condition (1) designates the minimal class **D'** as the result of the insertion. We define deletion of a tuple from an equivalence class in a similar manner.

Definition 3.4 Let U be a universe and let F be a set of functional dependencies over U. Define *deletion* of a tuple to be a function DEL from

BASES(F)/≡ x TUPLES into BASES(F)/≡ , such that, for every class **D** and every tuple t,

(1) DEL(**D**,t) is the largest class **D'** verifying conditions (2) and (3) below:

(2) **D'** < **D**

(3) D' $\not\models$ s, <u>for all s such that: D $\models$ s≤t.</u> ♦

It is shown in [LS87] that DEL(**D**,t) is always defined, that is, DEL is a total function. The reason why DEL(**D**,t) is required to be a maxiamal class is similar (or, rather, symmetric) to the reason why INS(**D**,t) is required to be a minimal class. The following proposition describes a basic property of deletion, namely, when deleting a tuple t, we must also delete all its refinements (i.e., all its supertuples).

Proposition 3.1

If D is a database and t is any tuple, then we have:

$T(DEL(D,t)) = T(D) - \{s \mid s \in T(D),\ D \models s \leq t\}$.

Proof see [LS87]

Example 3.2 Let U = {A,B,C,D} be a universe, let F = {B→D, C→D} be a set of functional dependencies, and let D = {ab , bc}. We have T(D) = {ab, bc, a, b, c}.

<u>Insertion</u>: We consider the insertion of the tuple cd in D. A representative of the class INS(**D**,cd) is the database D' = {ab, bc, cd}, so our semantics for insertion corresponds to the intuitive notion of adding a tuple to a database. Moreover, we have T(D') = {abd, bcd, ab, bc, bd, cd, ad, a, b, c, d}.

<u>Deletion:</u> We consider the deletion of the tuple ad from the database D' above. D' $\models$ abd ≤ ad because abd is a super-tuple of ad. Moreover, D' $\models$ ab ≤ abd because of the dependency B→D. We deduce from Proposition 3.1 above that the result of the deletion is the class **D"** such that T(D") = T(D')-{ab,abd, ad} = {bcd, bc, bd, cd, a, b, c, d}. A possible representative of this result is the databaseD" = {bc, cd, a}. ♦

It should be clear from definitions 3.3 and 3.4 that, in order to process insertion and deletion of tuples, all we need is a procedure for solving the following inference problems: Given a database D and tuples t and x,

(1) Decide whether D $\models$ t,

(2) Decide whether D $\models$ $t \leq x$

In other words, we have to solve precisely the *same* inference problems as in the case of queries! This is the reason why our decision procedure [LS87] strictly subsumes chase and weak chase, as we shall now explain.

3.3 Partition Semantics and Weak Instances.

Weak instances were first introduced as a means for discussing global satisfaction of a set of dependencies [M84] and has since been used in inferring missing information in a database state, discussing equivalence of database states, and defining Window Functions [MRW86]. Its semantics is captured by the well known chase procedure. The so-called weak-instance model seems to provide the right framework for querying universal scheme interfaces. However, there does not seem to be the flexibility to update over arbitrary schemes that there is to query over arbitrary schemes. In this paragraph, we compare the weak instance model, with our partition model. In the following, we denote by WChase a chase procedure in which a non distinguished variable can only be replaced by a distinguished variable and not by another non distinguished variable (this is what [MRW86] called 'null preserving chase'). We denote by $\rho(D)$ the tableau built from the database D by padding out the tuples of D with non distinguished variables. Finally, we denote by $\pi{\downarrow}_R(X)$ the set of all tuples in $\pi_R(X)$ containing no nulls (usually, $\pi{\downarrow}_R(X)$ is called *restricted projection* of X on R).

Theorem 3.1 Let $D=(\delta,F)$ be a database and let t be a tuple over a relation scheme R:

(1) D is consistent iff $\mathrm{WChase}_F(\rho(D))$ satisfies F.

(2) D implies t iff t is in $\pi{\downarrow}_R(\mathrm{Wchase}_F(\rho(D)))$. ♦

N.B. The terms "consistent" and "implies" have the sense of our model, described in Section 2 above, whereas the term "satisfies" has the sense of the relational model.

Proof see [LS87].

This theorem establishes the equivalence between our semantics and those of weak chase. However, our semantics is more general in two important ways (see [LS87] for more details). More precisely,

(a) As we noted earlier, relational functional dependencies can be interpreted in two ways in our model, namely, they can be interpreted as partial or total functions. If they are interpreted as partial functions, then our semantics is equivalent to those of Weak Chase, whereas, if they are interpreted as total functions, then our semantics is equivalent to Chase (ie to the weak instance semantics). Thus, in terms of expressive power, our model stricly subsumes the weak instance model. For the purposes of this paper, we have chosen to interpret functional dependencies as partial functions (see Def 2.4). Let us note in the passing that Chase and Weak Chase coincide over an important class of database schemes, namely independent schemes [MRW86]. For that specific class, our model and the weak instance model have the same expressive power.

(b) As we have seen earlier, the decision procedure for answering queries in our model can also be used for processing updates. We consider this to be an important advantage of our model with respect to the weak instance model.

4. SYNTACTIC PROCESSING OF QUERIES AND UPDATES

In the previous section, we have seen that the same decision procedure (one that determines whether $D \models t$ and whether $D \models t \leq x$) is sufficient for processing both queries *and* updates, in a universal scheme interface. Thus, all we have to do is to implement such a procedure, in order to obtain a universal scheme interface. And, in fact, such a procedure is given in [LS87] and it has been implemented in C [M87] (a prototype system is now running on a SUN workstation). However, this approach to implementing a universal scheme interface is, essentially, building a system from scratch.

In this paper, we propose a more pragmatic approach to the problem. Namely, rather than implementing a universal scheme interface from scratch, we propose to take advantage of existing DBMS technology as follows:

queries: Given elementary query Q/X=x, rather than computing the set of tuples

$$\{t \in dom(Q) \,/\, D \models t,\ D \models t \leq x\}$$

(which is the answer to the query), try to define a relational algebra expression that computes the same set of tuples, and let a traditional DBMS evaluate this expression.

<u>insertion:</u> Given a database D and a tuple t to be inserted in T(D), rather than computing INS(**D**,t), (which is the result of the insertion), try to define a relational transaction whose result is in INS(**D**,t), and let a traditional DBMS execute this transaction.

<u>deletion:</u> Given a database D and a tuple t to be deleted from T(D), rather than computing DEL(**D**,t), (which is the result of the deletion), try to define a relational transaction whose result is in DEL(**D**,t), and let a traditional DBMS execute this transaction.

In the remaining of this section, we show that this approach does, in fact work and this result constitutes the main contribution of the paper. The method that we propose for implementing a universal scheme interface can be decomposed into two parts, as follows:

DESIGN PHASE :

Given universe U and set of functional dependencies F,
(1) We associate U and F with a (uniquely defined) database scheme, called the *canonical scheme* of U and F.
(2) We associate every nonempty subset Q of U with three (uniquely defined) objects:

- A relational expression on the canonical scheme
- An insertion transaction on the canonical scheme
- A deletion transaction on the canonical scheme.

<u>Note</u>: By transaction, we mean a sequence of queries, insertions and deletions.

<u>RUN TIME</u> :

The data is stored according to the canonical scheme. User queries and updates are expressed in terms of attributes alone, and the user does not have to know which attributes are in which relations of the canonical scheme. A query or update, expressed in terms of a set of attributes Q, is processed as follows:

query: The relational expression associated, during the design phase, with Q is evaluated on the current database.

insertion: The insertion transaction associated, during the design phase, with Q is evoked and executed on the current database.

deletion: The deletion transaction associated, during the design phase, with Q is evoked and executed on the current database.

It is important to note that, following our approach, the overhead required for the implementation of a universal scheme interface, is a one-shot operation. Indeed, this overhead appears only during the design phase, when one must generate the canonical scheme, and the relational expression and transactions associated with each distinct set of attributes. During run-time, however, no particular overhead is required, and the processing of queries and updates is done using traditional DBMS techniques. Therefore, our approach of implementing a universal scheme interface can be seen as adding an upper layer on a traditional DBMS. This upper layer is currently being developped in our laboratory.

4.1 Computing answers through relational expressions

Throughout our discussions, we consider fixed a universe U and a set of functional dependencies F. A *relational expression* over U is any well formed expression whose operators are relational algebra operators and whose operands are relation schemes over U. Given a database D and a relational expression e over a universe U, we can *evaluate* e over D by substituting database relations for the schemes in e and performing the operations. We denote the result of the evaluation by eval(e,D). Unfortunately, the evaluation of a relational expression does not always yield true tuples. To see this, consider the database D=({ab,bc},$\emptyset$) and the expression e=AB$\bowtie$BC. We have eval(e,D)={abc} although D does not imply abc! This example motivates the following definition:

Definition 4.1 Let F be a set of functional dependencies over a universe U. A relational expression e over U is called *sound* with respect to F iff every tuple in eval(e,D) is implied by D, for all D in BASES(F). ♦

As we have stated earlier, we are interested in answering queries by evaluating relational expressions. So, suppose that we want to answer a query Q/X=x over a database D in BASES(F). Moreover, suppose a family Δ of relation schemes over U whose union $\cup\Delta$ contains Q, and whose join $\bowtie\Delta$ is a sound relational expression with respect to F. Then the expression $\sigma_{X=x}(\pi_Q(\bowtie\Delta))$, when evaluated over D, produces tuples over Q that are implied by D and satisfy the condition X=x. It follows that these tuples are in the answer of Q/X=x. Motivated by this discussion, let us define a family Δ of relational schemes to be a *context* of Q iff $Q \subseteq \cup\Delta$ and $\pi_X(\bowtie\Delta)$ is a sound relational expression. We denote by con(Q) the set of all contexts of Q. Let eval(Δ,D) denote the result of the evaluation of the expression $\sigma_{X=x}(\pi_Q(\bowtie\Delta))$ over database D. It follows from our previous discussion that

$$\bigcup_{\Delta\in \mathrm{con}(Q)} \mathrm{eval}(\Delta,D) \subseteq \alpha(Q/X{=}x,D)$$

We shall show shortly (Theorem 4.5 below) that the inclusion also holds in the opposite direction. So, in order to compute α, we need to generate the set of all contexts of Q. The following theorem is a step in that direction, as it helps identify contexts.

Theorem 4.1 Let R and S be relation schemes over U. If $R\cap S\rightarrow R$ is in F^+, or $R\cap S\rightarrow S$ is in F^+, then the expression $R\bowtie S$ is sound with respect to F. ♦

<u>NB</u> F^+ denotes the closure of F under Armstrong axioms.

Proof

Let X=R-S, Y=$R\cap S$, Z=S-R. Every tuple in eval($R\bowtie S$,D) is of the form xyz where xy is a tuple over R and yz is a tuple over S. As xy is true, and as $Y\rightarrow X$ is in F^+, we have that $I(y)\subseteq I(x)$ for every model I of D. Hence,

$$I(xyz) = I(x)\cap I(y)\cap I(z) = I(y)\cap I(z) = I(yz)$$

and, as yz is true, so is xyz. A similar reasoning applies in the symetric case. ♦

In view of this theorem, define relation schemes R and S to be ***neighbors*** (with respect to F) iff $R\cap S\rightarrow R$ is in F^+, or $R\cap S\rightarrow S$ is in F^+. Consider, for example, the following universe U and set of functional dependencies F, that we shall use as our running example:

U={A,B,C,D} , F = {B→D, C→D}

Then, the relation schemes AB and BD are neighbors, as $B \rightarrow BD$ is in F^+. Similarly, the relation schemes ABC and BCD are neighbors, as $BC \rightarrow BCD$ is in F^+. Using the concept of neighbors, we can generate all contexts of a given relation scheme, as described in the following theorem.

Theorem 4.2 Let Q be a relation scheme, then

(1) $\{Q\}$ is a context of Q.

(2) If Δ is a context of $Q \cup P$, then Δ is a context of Q

(3) If Δ is a context of Q, Δ' is a context of P, Q and P are neigbours, then $\Delta \cup \Delta'$ is a context of $Q \cup P$.

(4) Nothing else is a context of Q.

Proof See Appendix.

In our running example, if we let Q=AD and apply Theorem 4.1, we find

$$\text{con}(AD) = \{ \{AB,BD\}, \{AC,CD\}, \{ABC,BCD\}, \{AB, BC, CD\}, \{AD\}, \ldots \}$$

(Recall that con(Q) denotes the set of all contexts of Q).

4.2 The canonical scheme

We have seen so far how we could compute the answer of a query Q using the set of all contexts of Q. In this section, given a universe U and a set of functional dependencies F, we define a database scheme that we call canonical scheme and denote by can(U,F). This database scheme will be used to consider only "useful" contexts, as we shall see later.

In order to arrive at the definition of the canonical database scheme, we need some additional definitions and notations. First, define a family Δ of relational schemes to be ***complete*** iff $\forall X \in \Delta, \forall Y \subseteq X, Y \in \Delta$. For instance, in our running example, the family $\{AB,BC,A,B,C\}$ is complete. Second, Define a context Δ of Q to be a *trivial context* of Q if there is P such that $Q \subseteq P$ and $P \in \Delta$. Now, define a family Γ to be ***redundant*** iff there is X in Γ, and non-trivial context Δ of X such that $\Delta \subseteq \Gamma$. For example, in our running example, the family $\Gamma = \{ ABD,AB,BD \}$ is redundant, because $\{AB,BD\}$ is a non-trivial context of ABD. Intuitively, as relations over ABD can be decomposed losslessly, over AB and BD, the scheme ABD is not needed. An example of irredundant family is the following: {AB, BC, B}.

It is not difficult to see that the union of two complete and irredundant families is also complete and irredundant. It follows that the union of all complete and irredundant families is also complete and irredundant. In fact, this union is the greatest (with respect to set inclusion) complete and irredundant family over U and F. We call this family the *canonical scheme* over U and F and we denote it by can(U,F). For example, in our running example, the canonical scheme is {ABC, AB, AC, AD, BC, BD, CD, A, B, C, D}.

Theorem 4.3 Let U be a universe and F a set of functional dependencies. For every database D in BASES(F), there is a database D' over the canonical scheme which is equivalent to D.

Proof

This result comes from the completeness property of the canonical scheme. Indeed, for each relation scheme X which is not in the canonical scheme, there is a subset $\{X_1, X_2, ..., X_n\}$ of the canonical scheme such that $X_i \subseteq X$, for all i, and for all tuple t over X, the database {t} is equivalent to $\{\pi_{X1}(t), ..., \pi_{Xn}(t)\}$. More generally, we can show that the database {t} is equivalent to the database $\{\pi_Y(t)$, for $Y \subseteq X$, and $Y \in$ can(U,F)$\}$. So any database D in BASES(F) can be transformed in an equivalent database D' over the canonical scheme.

Finally, we need the notion of reduction of a relation scheme. Let X be a relation scheme. The *reduction* of X, denoted by red(X) is the family of all relation schemes Y such that:

(1) $Y \subseteq X$,

(2) $Y \rightarrow X$ is in F^+.

(3) There is no $Y' \subseteq Y$ such that $Y' \rightarrow X$ is in F^+.

For instance, in our running example, we have red(A) = {A}, red(BC) = {BC}, red(BD) = {B}, red(ABD) = {AB}.

The reductions of relation schemes have an interesting property, namely, they all belong to the canonical scheme, as stated in the following theorem.

Theorem 4.4 For every relation scheme X, we have

(1) $red(X) \subseteq can(U,F)$.

(2) $X \in can(U,F) \Leftrightarrow X \in red(X)$ ♦

proof

The canonical scheme can(U,F) is obtained by removing from 2^U all relation schemes that can be decomposed (losslessly) into two or more subschemes. As all Y in red(X) are minimal subsets of X, they cannot be split any further, so they are all in can(U,F).

We are now ready to define the implementation of queries, insertions, and deletions in a universal scheme interface.

4.3 Queries

Let U be a universe, let F be a set of functional dependencies, and let can(F,U) be the associated canonical scheme. Given a query q=(Q/X=x), we are looking for a relational expression e_q such that $\alpha(Q,D) = eval(e_q,D)$, for all D in BASES(F). We have the following theorem:

Theorem 4.5: Let q=(Q/X=x) be an elementary query over U.

Let $$e_q = \bigcup_{\substack{\Delta \in con(Q) \\ \Delta \subseteq can(U,F)}} \sigma_{X=x} (\pi_Q (\bowtie \Delta))$$

Then for every database D over the canonical scheme, we have:

$\alpha(q,D) = eval(e_q,D)$.

Proof

We have shown that $eval(e_q,D)$ is a subset of $\alpha(q,D)$. The converse comes from the fact that a tuple implied by a database D can always be computed, using joins and projections, from the tuples of the database. The contexts of a relation scheme Q are, by definition, all the sound join paths, that is, all join paths giving true tuples on Q. So every true tuple over Q is in the projection over Q of one of these paths. ♦

For instance, in our running example, consider the query q=(BC/B=b). We compute first the set of all contexts of BC and the canonical scheme associated to U and F :

$$\mathrm{con}(BC) = \{ \{BC\}, \{BD, CD\}, \{ABC\}, \{BCD\} \}$$

$$\mathrm{can}(U,F) = \{ABC, AB, AC, AD, BC, BD, CD, A, B, C, D\}.$$

It follows that the relational expression associated to q is

$$e_q = [\sigma_{B=b}\ \Pi_{BC}\ (BD \bowtie CD)] \cup [\sigma_{B=b}\ \Pi_{BC}\ (ABC)] \cup [\sigma_{B=b}(BC)] .$$

Several systems have been proposed, based on universal scheme interfaces, such as SYSTEM/U [KFGU84], or PIQUE [MRSSW87], allowing the user to query a relational database, without any knowledge of its logical structure. The PIQUE language, in particular, provides a very convenient way for querying universal scheme interfaces. However, two major problems still remain with such systems. First of all, none of them allows the user to *update* over arbitrary schemes. Second, these systems lack a precise semantics for data, at the tuple level. As a consequence, the system must still make a choice between different possible access paths. In PIQUE, this problem is delegated to the so-called "Object Based Generator" which is in charge to provide a relational expression for representing a relation scheme. This Object Based Generator does not take semantic information into account, such as functional dependencies. In conclusion, existing systems do not go far enough toward logical data independence which is the final goal of universal scheme interfaces. We believe that only a semantic approach, such as ours, can reach this goal.

In the following section, we present updates processing in universal scheme interfaces, which is the main contribution of our paper.

4.3 Updates

Let us call canonical transaction any mapping T that takes as arguments a database D over the canonical scheme and a tuple t, and returns a database T(D,t) over the canonical scheme. Such a mapping must be viewed as a sequence of queries, insertions and deletions on relation schemes of the canonical scheme. It is important to note that the tuple t can be *any* tuple. That is, t does not have to be over a relation scheme of the canonical scheme.

Insertion

Let D be a database on the canonical scheme, and let t be a tuple to be inserted in D. We are looking for a canonical transaction T_{INS} such that: T_{INS} (D,t) is in the class INS(**D**,t). We have the following theorem:

Theorem 4.6 Let D=(δ,F) be a database over the canonical scheme. Let t be a tuple over relation scheme Q (NB. Q is not necessarily in the canonical scheme). If INS(**D**,t) is defined, then the following canonical transaction leads to a database in INS(**D**,t).

T_{INS} (D,t) : For all X in can(U,F) such that $X \subseteq Q$.

| insert $\Pi_X(t)$ in δ(X). ♦

Proof

We have seen (in the proof of Theorem 4.4) that a tuple t over Q is semantically equivalent to the set of all tuples $\Pi_X(t)$ such that $X \subseteq Q$ and $X \in$ can(U,F). So, inserting all these tuples in D is semantically equivalent to inserting the tuple t itself.

For instance, in our running example, suppose that we want to insert the tuple abd. Moreover, suppose that the "current" database D is empty so that INS(**D**,abd) is defined. The corresponding canonical transaction is the following:

T_{INS}(D,abd) : insert the tuples ab, bd, ad, a, b and d in the corresponding relations.

Notice that we could avoid inserting the tuples ad, a, b and d because they are implied by ab and bd. Our system is expected to do these optimizations.

Deletions

Let D be a database on the canonical scheme, and let t be a tuple to be deleted from D. We are looking for a canonical transaction T_{DEL} such that: T_{DEL} (D,t) is in the class DEL(**D**,t). We have the following theorem:

Theorem 4.6 Let D be a database over the canonical scheme. Let t be a tuple over relation scheme Q (NB. Q is not necessarily in the canonical scheme). DEL(**D**,t) is always defined and the following canonical transaction leads to a database in DEL(**D**,t).

T_{DEL} (D,t) : For all X such that $Q \subseteq X$,

For all Y in red(X),

For all s in α(X/Q=t,D),

delete $\Pi_Y(s)$ from $\delta(Y)$, and

insert $\Pi_Z(s)$ in $\delta(Z)$, For all Z such that $Z \subseteq Y$ and $Z \neq Y$. ♦

Proof (sketch)

Using the semantics defined in Section 3, we must remove from the database all tuples t' such that D implies $t' \leq t$. All tuples in α(X/Q=t,D),for a superset X of Q, satisfy this condition. We could show that if s is a tuple over X, and if Y is a relation scheme in red(X), then $m(s)=m(\Pi_Y(s))$ for all models m of D. So, looking at Theorem 4.3, it is necessary and sufficient to remove all tuples $\Pi_Y(s)$ from the corresponding relations in order to delete s. Now, every (strict) subtuple of $\Pi_Y(s)$ is inserted, in order to capture the semantics of Section 3 (the maximality condition). Notice that all subsets Z of Y are in can(U,F), because Y is in can(U,F), from Theorem 4.3.

For instance, in our running example, suppose that we want to delete ad, from the database D (containing tuples ab, bc, and cd). Following Theorem 4.6, the corresponding transaction is T_{DEL}(D,ad):

for X=ADB, we have α(ADB/AD=ad,D) = {adb}, and red(ADB) = {AB}, so we

- delete ab from the relation over AB, and
- insert a and b in the corresponding relations;

for X=ADC, we have α(ADC/AD=ad,D) = Ø;

for X=ADBC, we have α(ADBC/AD=ad,D) = Ø.

We thus obtain a database containing the tuples a,b,bc, and cd. We can compare this result with the example 3.2 of Section 3.

5. CONCLUSIONS

We have used partition semantics in order to implement queries and updates on a universal scheme interface, through relational expressions and transactions on a traditional DBMS. Implementations of queries, based on the weak instance model, have been proposed in the past. However, proposals for the implementation of updates have only led to increased complexity without really soving the problem. We believe that it is in the implementation of updates that our method offers a simple solution.

The approach described in this paper is currently being implemented in the form of an upper layer for traditional DBMS's. At the same time, another important aspect of our approach is being investigated, namely, concurrency of transactions. We believe that our approach offers greater flexibility over the traditional approach, due to the fact that we are updating equivalence classes of databases, rather than specific databases. This means that we can relax the traditional definition of transaction equivalence, which requires two transactions to produce the *same* database in order to be equivalent. Instead, we define two transactions to be equivalent if they produce equivalent databases. The consequences of this definition on concurrency are now under investigation.

APPENDIX

proof of Theorem 4.2

The proof of theorem 4.2 can be divided into two parts:

validity : (1), (2), and (3)

completeness : (4).

Validity :

(1) is obvious.

(2) comes from the equality: $\Pi_X(\bowtie\Delta) = \Pi_X(\Pi_{X\cup Y}(\bowtie\Delta))$.

(3): Let D be a database, t a tuple in $\bowtie(\Delta\cup\Delta')$. t is the join of two tuples t_1 and t_2 respectively in $\bowtie(\Delta)$ and $\bowtie(\Delta')$. If $\Pi_X(t)=\Pi_X(t_1)$ and $\Pi_Y(t)=\Pi_Y(t_2)$ are in T(D), and X,Y are neighboors, then we can deduce that $\Pi_{X\cup Y}(t)$ is also in T(D), so $\Delta\cup\Delta'$ is a context of $X\cup Y$.

Completeness:

Notations :

- If Δ is a context, $|\Delta|$ is the union of all relation schemes in Δ. That is, $|\Delta| = \cup_{X\in\Delta} X$.

- If $t_1, t_2, \ldots, t_n$ are tuples, then $T_{t1\ t2 \ldots tn}$ is the tableau over U obtained by padding the the tuples $t_1, t_2, \ldots, t_n$ with distinguished null values.

- We associate to each attribute A_i of U, a distinguished symbol a_i of $dom(A_i)$, and for all $X = \{A_1,..., A_n\}$, r_X denotes the tuple $a_1 a_2 ... a_n$.
- If Δ is a context, and $\Delta = \{X1,...Xn\}$, T_Δ denotes the tableau $T r_{X1} r_{X2},... r_{Xn}$.

Lemma 1 : If Δ is a context of X, $\Delta' \subseteq \Delta$, and $X \subseteq |\Delta'|$, then Δ' is a context of X.

We assume that $\Delta = \Delta' \cup \{R_1\}$. We can generalize by induction, to any subset of Δ. R_1 can be split into two subschemes R_2 and R_3, such that $R_2 \subseteq |\Delta'|$ and $R_3 \cap |\Delta'| = \varnothing$. Let D be a database in BASES(F). We can assume that $D(R_3)=\varnothing$ because the attributes of R_3 do not appear in Δ', so $D(R_3)$ is independent from $\bowtie(\Delta')(D)$. Let t be a tuple in $\Pi_X(\bowtie\Delta')(D)$. Let $t2 = \Pi_{R2}(t)$, and t_3 be any tuple over R_3. Let $D' = D \cup \{t_2 t_3\}$. As X is included in Δ', $\Pi_X(t\ t_3) = \Pi_X(t)$. As Δ is a context of X, we have $\Pi_X(t_3) \in \Pi_X((\bowtie\Delta)\bowtie R_1)(D') \subseteq T(D')$, and so $\Pi_X(t)$ is in T(D'). As the symbols of the attributes of R_3 do not appear in t nor in D, we can conclude that $\Pi_X(t)$ is also in T(D). So, finally, Δ' is a context of X.

An important consequence of this lemma is the following: if Δ is a context of X, R_1 and R_2 are in Δ, and $R_1 \subseteq R_2$, then $\Delta - \{R_1\}$ is still a context of X.

This lemma allows us to consider only contexts Δ verifying the following properties:

1) $\forall R \in \Delta,\ X \not\subset |\Delta - \{R\}|$, and

2) $\forall Y,\ (X \subseteq Y \wedge D \in con(Y)) \Rightarrow X = Y$.

Indeed, If Δ is a context, we can deduce from Lemma 1 that the set Δ', obtained from Δ by removing all relation schemes R such that $X \subseteq |\Delta - \{R\}|$, is another context of X. Moreover, axioms 1 to 3 allow us to conclude that Δ is a context of X, knowing that Δ' is a context of X:

$\{R\} \in con(R)$ (From axiom (1)).

so $\{R\} \in con(\varnothing)$ (From axiom (2)).

$\varnothing$ and X are neighboors, so $\Delta' \cup \{X\} \in con(X)$. (From axiom (3)).

In the remaining of the proof, we shall consider only "reduced" contexts, that is contexts verifying properties 1) and 2) above.

Lemma 2 : for all Δ and X, we have:

$\Delta \in \mathrm{con}(X)$ iff $\Pi{\downarrow}_X($ WChase $_F(T_\Delta)) \neq \emptyset$.

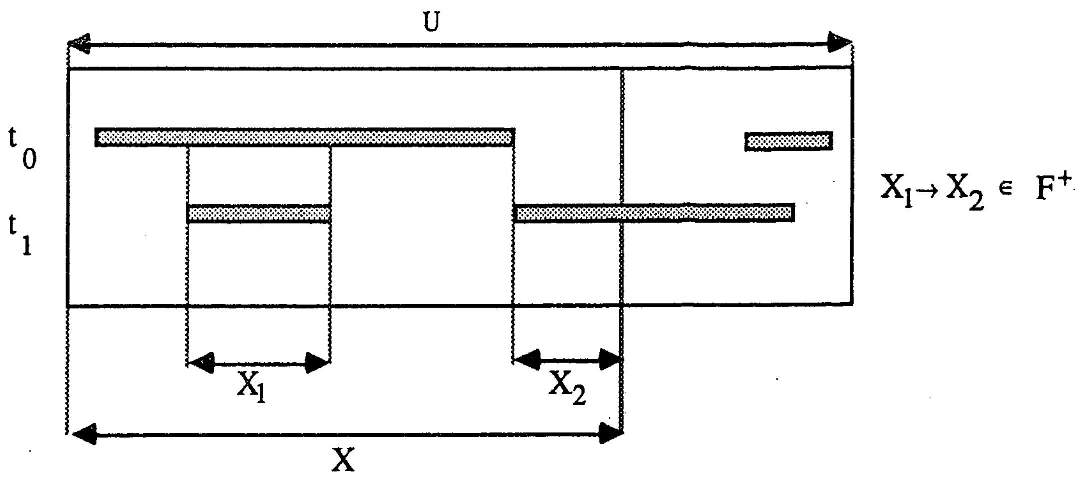
Proof

We know [LS87] that a tuple t over R is in T(d) iff t is in $\Pi{\downarrow}_R($ WChase $_F(T_d))$. If $\Delta = \{X_1, ..., X_n\}$ is a context of X, then the projection on X of the join of the tuples r_{X1} ... and r_{Xn} is true, by definition. So this tuple is in the restricted projection $\Pi{\downarrow}_X($ WChase $_F(T_\Delta))$, and this restricted projection is, thus, nonempty. Conversely, assume that the restricted projection is nonempty, then let d be a database over the given universe, and t a tuple in $(\bowtie\Delta)(d)$. The tuple t can be written as $t = t_1\ t_2\ ...\ t_n$, with t_{1i} in $\mathrm{dom}(X_i)$, for i=1,..,n. As Δ is a reduced context, t can only be obtained with the join $t_1\ t_2\ ...\ t_n$. (ie a join involving only a (strict) subset of $t_1\ t_2\ ...\ t_n$ could not be equal to t). At this point, we can establish a one-to-one correspondance between the tuples $t_1\ t_2\ ...\ t_n$ and the tuples r_{X1} ... and r_{Xn}. Now, as $\Pi{\downarrow}_X($ WChase $_F(T_\Delta))$ is not empty, the relation $\Pi{\downarrow}_X($ WChase $_F(T_{t1\ t2\ ...\ tn}))$ is not empty either. Moreover, this restricted projection can contain at most one element which is $\Pi_X(t)$ (by definition of $t_1\ t_2\ ...\ t_n$). So $\Pi_X(t)$ is in $\Pi{\downarrow}_X($ WChase $_F(T_{t1\ t2\ ...\ tn}))$ and so in $\Pi{\downarrow}_R($ WChase $_F(T_d))$ because $t_1\ t_2\ ...\ t_n$ are tuples of d. So we can conclude that $\Pi_X(t)$ is in T(d), and, finally, that Δ is a context of X.

Let us use these two lemmas in order to show the completeness of our axiom system: Let Δ be a (reduced) context. We know from Lemma 2 that $\Pi{\downarrow}_X($ WChase $_F(T_\Delta))$ is not empty. Two cases may appear at this time:

- $\Pi{\downarrow}_X(T_\Delta)$ is not empty. Then, X is included in an element of Δ. As Δ is a reduced context, we necessarily have $\Delta = \{Y\}$, with $X \subseteq Y$. We can generate such a context with axioms 1 and 2.

- Otherwize, we can detail the steps of the chase computation. In order to do this, we denote by T^0 the initial tableau , and by T^{n+1} the tableau obtained by applying the chase rule on T^n. Let i be the least integer such that $\Pi{\downarrow}_X(T^i)$ is empty and $\Pi{\downarrow}_X(T^{i+1})$ is not empty. The chase rule applied in order to obtain T^{i+1} from T^i consist in the "join" of two tuples t_0 and t_1, as showed in the following figure.



We obtain the tableau T^{i+1} by replacing $t_0[X_2]$ by $t_1[X_2]$ in T^i, transforming the tuple t_0 into a tuple defined (at least) over X. Let us note $X_0 = X - X_2$. We know that Δ is a context of X_0 because X_0 is included in X and Δ is a context of X. We also know that Δ is a context of X_1X_2 for the same reason. Moreover, X_0 and X_1X_2 are neigbours because $X_0 \cap X_1X_2 = X_1$ and $X_1 \to X_1X_2$ is in F^+. So we can use Axiom 3) and the problem of deriving that Δ is a context of X is reduced to the two sub_problems : Δ is a context of X_0, and Δ is a context of X_1X_2. As X_0 and X_1X_2 are strictly included in X, these two subproblems are simpler than the original one, and we can conclude that every context Δ of X will be generated using our three axioms. So our axiom system is complete.

BIBLIOGRAPHY

[CKS86] Cosmadakis S., Kanellakis P.C., Spyratos N., Partition Semantics for Relations, ACM PODS, March 1985.

[KFGU84] Korth H.F., Kuper G., Feigenbaum J., VanGelger A., Ullman J.D., System U: A Database System Based on the Universal Relation Assumption. ACM TODS, 9:3, September 1984.

[LS87] Lécluse C., Spyratos N. Updating Weak Instances Using Partition Semantics, Rapport de Recherche LRI, No 364, July 1987.

[M84] Mendelzon, AO. Database states and their tableaux, ACM TODS 9:2 p264-282, 1984.

[M87] Mancuso F., Implémentation d'un modèle de données ensembliste. Rapport de stage de DEA, Université de Paris Sud, Orsay, Septembre 1987.

[MRW86] Maier D., Rozenshtein D., Warren DS., Window functions, in Advances in Computer Research 3, p213-246, JAI Press, London.

[MRSSW87] Maier D., Rozenshtein D., Salveter S., Stein J., Warren D., Pique: A Relationnal Query Language Without Relations, in Information Systems, Vol 12, No 3, 1987.

[S84] Spyratos N., The partition model - a deductive database model. ACM TODS, March 1987.

[S83] Sagiv Y., A Characterisation of Globally Consistent Databases and their Correct Acess Path. ACM TODS 8:2, p266-286, 1983.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique